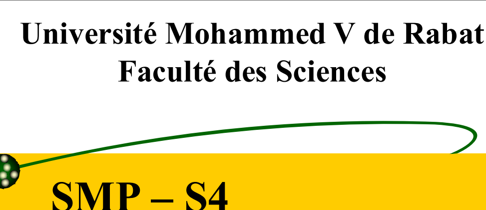


Université Mohammed V de Rabat
Faculté des Sciences

SMP – S4
Module: Informatique

Langage C

M. HIMMI

Printemps 2015

Présentation intuitive du langage C
1 - Premier programme en C

```
main()
{
      printf ("bonjour");
   }
```

```
bonjour
```

- ***main()*** précise que ce qui suit est le programme principal.
- Il est délimité par les accolades '***{***' et '***}***'
- Dans cet exemple nous avons une seule instruction ***printf(…);***
- Le point-virgule qui termine cette instruction est **obligatoire**
- Toute instruction simple est suivie d'un point-virgule

Présentation intuitive du langage C
2 – Le caractère de fin de ligne

```
main()
{
  printf ("bonjour\n");
 printf ("tout le monde");
}
```

```
bonjour
tout le monde
```

- Cette fois le programme principal comporte 2 instructions:
- La première instruction contient la notation '**\n**' : c'est le caractère de fin de ligne. Il provoque un passage à la ligne suivante.
- Le langage C prévoit une notation de ce type pour différents caractères dits: caractères de contrôle

---

Présentation intuitive du langage C
3 – Les variables et leurs types

```
main()
{
  int n;  n=10;
  printf ("valeur %d",n);
}
```

```
valeur 10
```

- La première instruction est une déclaration. Elle précise que la variable nommée ***n*** est de type ***int*** (entier).
- La deuxième instruction est une affectation. Elle place la valeur ***10*** dans la variable ***n***.
- La troisième instruction demande d'afficher la valeur de la variable ***n*** suivant le format "***valeur %d***"

Présentation intuitive du langage C
4 – Le type caractère et le code format %c

```
main()
{
  char x;   x= 'e';
  printf ("lettre = %c",x);
}
```

```
lettre = e
```

- Déclaration que ***x*** est une variable de type ***char*** (caractère).
- La notation ***'e'*** désigne une constante caractère (ne pas confondre avec ***"e"*** qui désigne une chaîne de caractères constante ne contenant qu'un seul caractère).
- Le code format ***%c*** affiche une valeur de type caractère quelle qu'elle soit.

Présentation intuitive du langage C
5 – Afficher un seul caractère: putchar()

```
main()
{
  char x;
  x= 'e';
  printf ("lettre = ");
  putchar(x);
}
```

```
#include <stdio.h>
main()
{  char x;
  x= 'e';
  printf ("lettre = ");
  putchar(x);
}
```

- L'instruction ***putchar(x);*** demande d'afficher la valeur du contenu de ***x***
- Ce programme ne fonctionne pas, il est nécessaire de la faire précéder par une directive ***#include***
- ***putchar()*** est une macro-instruction qui doit être incorporée au code source avant compilation
- L'instruction ***#include <stdio.h>*** demande d'insérer le contenu du fichier ***stdio.h*** au programme

---

Présentation intuitive du langage C
6 – Lecture d'information au clavier

```
#include <stdio.h>
main()
{  char c;
  printf ("donnez un caractère : ");
  c=getchar();
  printf("merci pour %c",c);
}
```

```
donnez un caractère : w
merci pour w
```

- ***getchar()*** est aussi une macro mais ne possède aucun argument.
- Les parenthèses permettent de reconnaître que ***getchar*** est une fonction et pas une variable.
- ***getchar*** fournit une valeur en retour: le caractère lu au clavier, c'est cette valeur qui est affectée à la variable ***c***

Présentation intuitive du langage C
7 – Lecture d'information au clavier

```
#include <stdio.h>
main()
{  int n, p ;
  printf ("donnez deux nombres : ");
  scanf("%d%d", &n; &p );
  printf("leur somme est %d" ,n+p);
}
```

```
donnez deux nombres : 8 15
leur somme est 23
```

- Comme ***printf***, ***scanf*** possède en argument **un format** sous forme de chaîne de caractère ***"%d%d"*** qui correspond à deux valeurs entières et **une liste** indiquant les adresses des variables dont on veut récupérer ces valeurs.
- ***&*** est un opérateur signifiant adresse de

Présentation intuitive du langage C
8 – faire des boucles…

```
#include <stdio.h>
main()
{   int i, n ;
  printf ("Bonjour\n");
  printf ("je vais vous calculer 3 carrés\n");
  for ( i=1; i <= 3 ; i++)
       { printf("donnez un nombre entier : ");
         scanf("%d",&n);
         printf("son carré est : %d\n",n*n);
       }
  printf("au revoir");
}
```

```
Bonjour
je vais vous calculer 3 carrés
donnez un nombre entier : 3
son carré est 9
donnez un nombre entier : 8
son carré est 64
donnez un nombre entier : 5
son carré est 25
au revoir
```

- La répétition est réalisée par ***for (i=1; i<=3 ; i++)*** suivie par un ensemble d'instructions délimité par {}que l'on nomme bloc
- Ce qui signifie: répéter le bloc d'instructions qui suit en respectant les consignes suivantes:

Présentation intuitive du langage C
8 – faire des boucles…consignes

```
#include <stdio.h>
main()
{   int i, n ;
  printf ("Bonjour\n");
  printf ("je vais vous calculer 3 carrés\n");
  for ( i=1; i <= 3 ; i++)
       { printf("donnez un nombre entier : ");
         scanf("%d",&n);
         printf("son carré est : %d\n",n*n);
       }
  printf("au revoir");
}
```

```
Bonjour
je vais vous calculer 3 carrés
donnez un nombre entier : 3
son carré est 9
donnez un nombre entier : 8
son carré est 64
donnez un nombre entier : 5
son carré est 25
au revoir
```

•Avant de commencer cette répétition, réaliser ***i=1***

•A chaque nouvelle exécution tester la condition ***i<=3***. Si oui exécuter le bloc sinon passer à l'instruction suivant le bloc

•A la fin de chaque exécution du bloc, réaliser ***i++***

Présentation intuitive du langage C
9 – La directive *#define*

```
#include <stdio.h>
main()
{   int i, n ;
  printf ("Bonjour\n");
  printf ("je vais vous calculer 3 carrés\n");
  for ( i=1; i <= 3 ; i++)
       { printf("donnez un nombre entier : ");
         scanf("%d",&n);
         printf("son carré est : %d\n",n*n);
       }
  printf("au revoir");
}
```

• Dans cet exemple on a fait apparaître la constante ***3*** à deux reprises. Cette manière de faire ne facilite pas les modifications dans un programme…

• Une façon classique d'améliorer la situation consiste à placer cette valeur dans une variable ***nc***:

Présentation intuitive du langage C
9 – La directive #define

```
#include <stdio.h>
main()
{  int i, n , nc=5;
  printf ("Bonjour\n");
  printf ("je vais vous calculer %d
carrés\n",nc);
  for ( i=1; i <= nc ; i++)
       { printf("donnez un nombre entier : ");
         scanf("%d",&n);
         printf("son carré est : %d\n",n*n);
       }
  printf("au revoir");
}
```

• Une manière plus élégante est l'utilisation de la directive ***#define***.

•Le programme devient:

Présentation intuitive du langage C
9 – La directive #define

```
#include <stdio.h>
#define NC 3
main()
{
  int i, n ;
  printf ("Bonjour\n");
  printf ("je vais vous calculer %d
carrés\n",NC);
  for ( i=1; i <= NC ; i++)
       { printf("donnez un nombre entier : ");
         scanf("%d",&n);
         printf("son carré est : %d\n",n*n);
       }
  printf("au revoir");
}
```

• La directive ***#define NC 3*** permet de remplacer, partout où il apparaît, NC par 3 (sauf dans les chaînes de caractères)

Avantage éviter la modification involontaire de la valeur ***NC***:

NC=NC+1 deviendrai ***3=3+1*** et provoquerai une erreur…

Présentation intuitive du langage C
10 – faire des choix…

```
#include <stdio.h>
main()
{   int a, b , q , r ;
    printf ("donnez deux entiers :");
    scanf("%d%d,&a,&b);
       if (b!=0)
    { q = a / b ;
      r = a % b ;
      printf ("division de %d pa %d\n",a,b);
      printf("       %d = %d * %d + %d", a ,
b , q , r );
    }
else printf ( "diviseur nul");
}
```

```
donnez deux entiers : 37 8
division de 37 par 8
    37 = 8 * 4 + 5
```

```
donnez deux entiers : 25 0
diviseur nul
```

• ***if*** réalise un choix basé sur la condition ***b!=0*** ***s***i elle est vrai on exécute le bloc sinon on exécute l'instruction située après ***else***.

La programmation

• Un programme est une suite d'instructions à destination d'une machine.

• Pour qu'une machine puisse exécuter un programme, il faut que celui-ci soit en langage machine qui se compose de deux symboles le 0 et le 1.

• Pour créer des programmes il faut programmer… c'est-à-dire écrire une **suite d'instructions** à destination de la machine.

• Il existe de nombreux langage de programmation comme le BASIC, le FORTRAN, le Pascal, le C … Ils servent à rendre humainement possible l'écriture de programmes.

• Le programmeur écrit des instructions proches du langage humain et à l'aide d'un interpréteur ou d'un compilateur, ce code est traduit en langage machine.

La programmation

Haut niveau ↑	Langage humain
	Langage de programmation haut niveau
	Langage assembleur
Bas niveau	Langage machine

La programmation

• Un programme écrit dans un langage de haut niveau est un fichier texte comportant des **instructions** et des **mots clés** proches du langage humain: **fichier source** .

• Pour être exécuté, un programme écrit dans un langage de haut niveau doit être **interprété** ou **compilé**.

• Il existe différents interpréteurs et compilateurs pour chaque langage selon la plateforme et le système d'exploitation.

• Le résultat de la compilation est le "*module objet*". Il est rangé dans un fichier ayant l'extension ***obj***.

Présentation du langage C

•**Caractéristiques:**

•**Modulaire**: Permet de découper une application en modules qui peuvent être compilés séparément.

•**Structuré:** Traite les tâches en les mettant dans des blocs.

•**Efficace:** Possède de grandes possibilités de contrôle de la machine et génère un code compact et rapide.

•**Portable:** Permet d'utiliser le même code source sur d'autres type de machines simplement en le recompilant

•**Extensible:** Animé par des bibliothèques de fonctions qui enrichissent le langage.

•**Souple et permissif:** Hormis la syntaxe, peu de vérifications et d'interdits ce qui peut poser des problèmes …

Présentation du langage C

Historique

• C a été créer en 1972 dans les 'Bell Laboratories' par ***Dennis M. Ritchie***.

• En 1978 publication de ***'The C Programming Language'*** par Brian W. Kernighan et Dennis M. Ritchie

• En 1983, l'ANSI commence le processus de normalisation du langage C. Le résultat était le standard ANSI-C.

• En 1988: deuxième édition du livre ***'The C Programming Language'***, qui respecte le standard ANSI-C. Elle est devenue la référence des programmeurs en C.

Présentation du langage C

Création d'un Programme:

Classiquement 3 étapes:

• **L'édition:** *Saisie et modification du texte du programme*

• **La compilation:** *Traduire en langage machine le texte constituant le programme, le module objet est rangé dans un fichier ayant l'extension obj.*

• **L'édition de liens:** *incorporation des modules objets par simple recopie de fichiers obj ou recherche dans un fichier lib qui regroupe plusieurs modules objets. Le résultat de l'édition des liens est un "exécutable". Il est rangé dans un fichier d'extension exe*

Présentation du langage C

Création d'un Programme:

Pour obtenir un exécutable il faut **réussir** successivement:

1. *la saisie les instructions des différents modules…*
2. *la compilation de tous les modules faisant partie du programme..*
3. *Le lien des différents modules objets pour obtenir l'exécutable.*

Fichiers en-tête

graphics.h
/* declarations */

/* declarations */

. . .

Programme principal

```
#include . . .
#include . . .
#include <graphics.h>
#include <math.h>

main (void)
{
. . .
}
```

Compiler

code compilé de la fonction 'main'

Bibliothèques précompilées

graphics.lib

maths.lib

. . .

Linker

Programme exécutable

.EXE

Présentation du langage C

Création d'un Programme:

On a deux possibilités:

1. Soit on récupère chacun de ces trois programmes séparément. C'est la méthode la plus compliquée, mais elle fonctionne..
2. Soit on utilise un programme "trois-en-un" qui combine l'éditeur de texte, le compilateur et l'éditeur des liens. Ces programmes sont appelés "Environnements de Développement Intégré" ou IDE

Présentation du langage C

Les Environnements de Développement Intégré (IDE) permettent de mettre en œuvre les étapes de développement d'un programme: édition, compilation, édition des liens, exécution, gestion de projets, mise au point..)

• Il existe plusieurs IDE disponibles gratuitement.

• Les IDE vous permettent de programmer sans problèmes. Certains sont plus complets au niveau des options, d'autres un peu plus intuitifs à utiliser, mais dans tous les cas les programmes que vous créerez seront les mêmes quel que soit l'IDE que vous utilisez.

Présentation du langage C

Exemples d'Environnements de Développement Intégré:

• **Qt Creator**	**www.qt.io/download/**
• **Code::Blocks**	**www.codeblocks.org/**
• **Microsoft Visual Studio**	**www.visualstudio.com/**
• **Gcc**	**gcc.gnu.org/**

Structure d'un programme C

Un programme C est composé de:

- **Directives du préprocesseur**
- **Déclarations et définitions**
- **Fonctions**
- **Des commentaires**

Structure d'un programme C

Directives du préprocesseur:

Elles permettent d'effectuer des manipulations sur le texte du programme source avant la compilation:

- inclusion de fichiers
- substitutions
- macros
- compilation conditionnelle

Une directive du préprocesseur est une ligne de programme source commençant par le caractère dièse '#'.

Le préprocesseur est appelé automatiquement, en tout premier, par la commande le compilateur

Structure d'un programme C

Déclarations et définitions:

• La déclaration d'un objet donne simplement ses caractéristiques au compilateur

• La définition d'un objet déclare cet objet et réserve effectivement l'espace mémoire pour cet objet

Structure d'un programme C

Fonctions:

• Ce sont des sous-programmes dont les instructions définissent un traitement sur des variables.

• Un programme est composé d'une ou plusieurs fonctions dont l'une **doit** s'appeler ***main,*** stockées dans un ou plusieurs fichiers.

•Une fonction est constituée de:

- **entête** : type et identifiant de la fonction suivis d'une liste d'arguments entre parenthèses
- **instruction composée** constituant le corps de la fonction.

Structure d'un programme C

Commentaires:

• Ce sont des textes explicatifs destinés aux lecteurs du programme et éliminés par le préprocesseur. Ils sont formés de caractères quelconques placés entre les symboles **/*** et ***/**

• Ils ne doivent pas être imbriqués et peuvent apparaître en tout point d'un programme.

```
/* programme de calcul de racines carrées */
/* Commentaire fantaisiste &ç#{<>]?%!!!!! @ */
/*  ===================  *
 *  commentaire s'étendant      *
 *   sur plusieurs lignes          *
 *  du programme source         *
 *  ===================  */
```

Pour ignorer une partie de programme il est préférable d'utiliser une directive du préprocesseur ***#if ()*** ... ***#endif***

Exemple de programme C

Code	Commentaire
`#include <stdio.h>`	Directives de compilation
`#define PI 3.14159`	
`main()`	Programme principal
`{`	Début bloc
`float rayon, surface;`	Déclarations
`float calcul(float rayon);`	
`printf("Rayon = ? ");`	Appel fonction écrire
`scanf("%f", &rayon);`	Appel fonction lire
`surface = calcul(rayon);`	Appel fonction calcul
`printf("Surface = %f\n", surface);`	Appel fonction écrire
`}`	Fin de bloc
`float calcul(float r)`	Définition fonction
`{`	
`float a;`	Déclarations locale
`a = PI * r * r;`	affectation
`return(a);`	Retour de la fonction
`}`	

Règles générales d'écriture en C

Les identifiants:

• Les identifiants sont les **noms** des objets (fonctions, variables, constantes, …), ce sont des suites de lettres et/ou de chiffres.

• Le premier caractère est **obligatoirement** une lettre. Le caractère souligné '_' est considéré comme une lettre.

• L'ensemble des symboles utilisables est donc:

0, 1, 2, ..., 9, A, B, ..., Z, _, a, b, ..., z

• Le C **distingue** les minuscules des majuscules.
Exemple : var Var VaR VAR sont des identifiants valides et tous **différents**.

• La longueur des identifiants n'est pas limitée, mais C distingue **seulement** les 31 premiers caractères

Règles générales d'écriture en C

Les identifiants: Exemples

Identifiants corrects	Identifiants incorrects
nom1	1nom
nom_2	nom.2
_nom_3	-nom-3
Nom_de_variable	Nom de variable
deuxieme_choix	deuxième_choix
mot_francais	mot_français

Règles générales d'écriture en C

Les mots clés: Ce sont des mots réservés par le langage C et qui ne peuvent pas être affectés à de identifiants

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Règles générales d'écriture en C

Instructions:
Une instruction est :

- soit une instruction simple,
- soit instruction composée ou bloc.

Une **instruction simple** est:
soit une instruction de contrôle,
soit une expression suivie de "***;***"

Une instruction **composée** ou **bloc** est:

- une accolade ouvrante "***{***"
- une liste de définitions locales au bloc (optionnelle)
- une suite d'instructions
- une accolade fermante "***}***".

Règles générales d'écriture en C

Les séparateurs:

• Deux identifiants successifs, entre lesquels il n'y a aucun operateur, doivent être séparés par un **espace** ou une **fin de ligne.**

On doit écrire int x,y
et non intx,y

Mais on peut écrire indifféremment: int n,compte,total,p
Ou plus clairement: int n, compte, total, p

Règles générales d'écriture en C

Le format libre:

• Le langage C autorise une "mise en page" libre.

• Les fins de ligne ne joue pas de rôle particulier si ce n'est celui de séparateur

• Une instruction peut s'étendre sur plusieurs lignes et une ligne peut contenir plusieurs instructions

• Cette liberté peut aboutir, si on ne fait pas attention, à des programmes peu lisibles:

```
main() { int i, n ;printf ("Bonjour\n"); printf (
"je vais vous calculer 3 carrés\n"); for ( i=1; i <=
 3 ; i++){ printf("donnez un nombre entier : ");scanf("%d"
,&n); printf("son carré est : %d\n",n*n); } printf("au revoir");}
```

Règles générales d'écriture en C

Respecter un certain nombres de règles de présentation :

- Ne pas placer plusieurs instructions sur une même ligne.
- Utiliser des identifiants significatifs.
- Réserver les identifiants en majuscules pour le préprocesseur.
- Faire ressortir la structure syntaxique du programme.
- Mettre une ligne entre les déclarations et les instructions.
- Une accolade fermante est seule sur une ligne et fait référence, par sa position horizontale, au début du bloc qu'elle ferme.
- Aérer les lignes de programme en entourant par exemple les opérateurs avec des espaces.
- Commenter les listings mais éviter les commentaires triviaux

Types et variables

Justification de la notion de type:

Toute information quelle que soit sa nature est codée sous forme binaire. Il ne suffit pas de connaître le contenu d'une zone pour lui attribuer une signification.

En C, une variable se caractérise à partir de son type.

La notion de type sert à:

- définir le domaine de valeurs (taille en mémoire et représentation machine)
- définir les opérations possibles sur cette variable

Types et variables

Types de base 1/2:

Ce sont les types prédéfinis. Ils sont au nombre de six :

void : c'est le type vide. Il est surtout utilisé pour préciser les fonctions sans argument ou sans retour.

int : c'est le type entier. Il se décline avec des qualificatifs pour préciser sa taille (***long ou short***), et le fait qu'il soit que positif (***unsigned***) ou positif et négatif (***signed***).
Par défaut ***signed*** est appliqué

char : ce type représente un entier sur huit bits. C'est le support des caractères codés en ASCII.
Comme le type ***int*** le type ***char*** peut être qualifié de ***signed*** ou ***unsigned***.

Types et variables

Types de base 2/2:

float : ce type sert pour les calculs avec des parties décimales.

double : c'est un type qui permet de représenter des valeurs ayant une partie décimale avec une plus grande précision que le type *float*. ce type est le plus courant pour représenter des valeurs avec parties décimales.

long double : ce type permet de représenter des nombres avec parties décimales qui nécessitent une très grande précision.

➢ **Remarque: Il n'y a pas de type booléen !**

Types et variables

Remarques

Avant de pouvoir déclarer une variable numérique, nous devons nous intéresser à deux caractéristiques de son type:

➢ le domaine des valeurs admissibles
➢ l'occupation mémoire nécessaire

▪ l'occupation mémoire qu'occupent les différents types dépend de la machine sur laquelle est implanté le compilateur.

▪Les fichiers d'en-tête *limits.h* et *float.h* contiennent des constantes symboliques qui précisent les tailles et les valeurs limites des entiers et des flottants.

▪Les tableaux suivants donnent une idée de la taille mémoire sur une machine 32 Bits :

Types et variables

Type	Description	Min	Max	Taille (octets)
char	caractère	-128	127	1
int	entier standard	-32 768	32 767	2
long	entier long	-2 147 483 648	2 147 483 647	4

Si on ajoute le préfixe ***unsigned*** les domaines des variables sont déplacés comme suit:

Type	Description	Min	Max	Taille (octets)
unsigned char	caractère	0	255	1
unsigned int	entier positif	0	65535	2
unsigned long	entier long	0	4 294 967 295	4

Types et variables

Types décimaux:

Type	Min	Max	Taille	Précision
float	$1.17 * 10^{-38}$	$3.4 * 10^{38}$	4	$1.19 * 10^{-7}$
double	$2.22 * 10^{-308}$	$1.79 * 10^{308}$	8	$2.22 * 10^{-16}$
long double	$3.36 * 10^{-4932}$	$1.18 * 10^{4932}$	16	$1.08 * 10^{-19}$

➢ Le type ***long double*** est récent et permet de représenter des nombres avec parties décimales sur une très grande précision, si la machine le permet.

Types et variables

Déclaration de variables de type simple:

<Type> <NomVar1>,<NomVar2>,...,<NomVarN>;

Exemples:

int compteur, X, Y, q_livree;
float hauteur, largeur;
double masse_atomique;
char touche;

Types et variables

Initialisation de variables:

- C'est l'affectation d'une valeur lors de la déclaration.
- Il suffit de faire suivre la définition du signe = et de la valeur que l'on veut voir attribuée à la variable.

int compteur = 0, X, Y, q_livree = 0;
float hauteur = 2.5, largeur = 1.2;
double masse_atomique ;
char touche = 'c';

- On utilise l'attribut ***const*** pour indiquer que la valeur d'une variable ne change pas au cours d'un programme:

const int MAX = 767;
const double e = 2.71828182845905;
const char NEWLINE = '\n';

Constantes ou type automatique

Chaque constante a une valeur et un type

Les constantes et les expressions constantes sont évaluées à la compilation.

L'expression ***1 + 2 * 3*** est traduite comme l'expression entière ***7***

Les constantes entières peuvent être exprimées sous les formes:

- Décimale: 1234
- Octale : 0177
- Hexadécimale : 0x1Bf 0XF2a

L'entier trop grand pour un ***int*** est considéré comme du type ***long***

On peut imposer à une constante entière d'être du type :

- *long* : 123 l 123L
- *unsigned* : 3456U
- *unsigned long* : 78UL

Constantes ou type automatique

Chaque constante a une valeur et un type

Les constantes réelles sont, par défaut, de type ***double***.

- Elles peuvent être exprimées sous les formes:

	Décimale :	1234.56	4.	-.3456
ou	Scientifique :	1.23e-12	12.3456E4	

- Elles peuvent aussi inclure un suffixe :

f ou ***F*** : impose le type ***float*** à la constante:
Exemple : 1.23f

l ou *L* : impose le type ***long double***
Exemple : 123.4567E45L

Constantes ou type automatique

Chaque constante a une valeur et un type

Les Constantes caractères sont écrites entre apostrophes.

- Elles peuvent être exprimées sous les formes suivantes :

Normale : **'a'**
Octale : **'\007'**
Hexadécimale : **'\x1b'**
Ou symbolique :
'\n' saut de ligne (LF)
'\a' bip
'\t' tabulation horizontale
'\b' retour arrière
…

Opérateurs et expressions

C dispose d'un grand nombre d'opérateurs:

- Opérateurs classiques (arithmétique, relationnels, logique)
- Opérateurs originaux d'affectation et d'incrémentation
- Opérateurs moins classiques (manipulation de bits)

Opérateurs et expressions

Opérateurs arithmétiques:

Type	Opérateur	Rôle
Binaire	+	Somme
Binaire	-	Différence
Binaire	*	Produit
Binaire	/	Quotient
Binaire	%	Reste de division (modulo)
Unaire	-	Opposé

Opérateurs et expressions

Remarques:

➢ Il n'existe pas d'opérateur d'élévation à la puissance !

➢ Les opérateurs binaire ne sont définis que lorsque leurs deux opérandes sont du même type.

➢ On peut écrire des expressions mixtes dans lesquelles interviennent des opérandes de types différents. L'évaluation de l'expression nécessite dans ce cas une **conversion d'ajustement de type**.

➢ La conversion d'ajustement de type suit une hiérarchie qui permet de ne pas altérer la valeur initiale:

int → long → float → double → long double

➢ Pour le type ***char,*** toute valeur apparaissant dans une expression est **d'abord** convertie en ***int*** .

Opérateurs et expressions

Priorités relatives des opérateurs arithmétiques:

- L'opérateur unaire – a la plus grande priorité
- On trouve ensuite et à un même niveau *******, */*, *%*
- En dernier niveau les opérateurs binaires + et –

➢Si les priorités sont identiques on calcule **de gauche à droite**

➢Les parenthèses permettent d'outrepasser ces règles

Exemples:

*a + b * c*	→	*a + (b * c)*
*a * b + c % d*	→	*(a * b) + (c % d)*
- c % d	→	*(- c) % d*
- a + c % d	→	*(- a) + (c % d)*
- a / - b + c	→	*((- a) / (- b))+ c*
- a / - (b + c)	→	*(- a) / (- (b + c))*

Opérateurs et expressions

Opérateurs relationnels:

Le C se distingue des autres langages par deux points:

1. Le résultat de la comparaison est un entier valant:
   0 si le résultat de la comparaison est faux
   1 si le résultat de la comparaison est vrai
2. Les expressions comparées peuvent être d'un type de base quelconque et sont soumises aux règles de conversion. On ne compare que des expressions de type numérique.

➢ Les opérateurs relationnels sont moins prioritaires que les opérateurs arithmétiques.

Opérateurs et expressions

Opérateurs relationnels:

Opérateur	Rôle
<	inférieur à
<=	inférieur ou égal à
>	supérieur à
>=	supérieur ou égal à
==	égal à
!=	Différent de

➢Les opérateurs <, <=, >, >= ont la même priorité.

➢Les opérateurs == et *!=* ont la même priorité mais celle-ci est inférieure aux premiers

Opérateurs et expressions

Opérateurs logiques:
C dispose des trois opérateurs logiques classiques et , ou, non

Comme pour opérateurs relationnels
vrai est représentée par *1*
et ***faux* par *0***

- Le résultat d'une comparaison est numérique de type *int*
- Ces opérateurs acceptent n'importe quel opérande numérique en utilisant les règles de conversion implicites

0 correspond à ***faux***
Toute autres valeur correspond à ***vrai***

Opérateurs et expressions

Opérateurs logiques:

Opérande 1	Opérateur	Opérande 2	Résultat
0	&&	0	0
0	&&	non nul	0
non nul	&&	0	0
non nul	&&	non nul	1
0	||	0	0
0	||	non nul	1
non nul	||	0	1
non nul	||	non nul	1
	!	0	1
	!	non nul	0

Opérateurs et expressions

Opérateurs logiques:

- L'opérateur ***!*** a une priorité **supérieure** à celle de tous les opérateurs arithmétiques binaires et aux opérateurs relationnels
- Le contraire de ***a==b*** est ***!(a==b)*** (avec parenthèses)
- L'opérateur || est moins prioritaire que ***&&***. Tous deux sont de priorité **inférieure** à celle de tous les opérateurs arithmétiques binaires et aux opérateurs relationnels.

➢ L'opérande de droite n'est évalué que si la connaissance de sa valeur est indispensable. Dans l'expression ***a<b && c<d***

Si ***a<b*** est *faux,* ***c<d*** ne sera pas évaluée puisque de toute façon le résultat aura la valeur *faux*

Opérateurs et expressions

Opérateur d'affectation simple:

- Contrairement à d'autres langages ***i=5*** est une expression qui:
  réalise une action: l'affectation de la valeur ***5*** à ***i***
  et
  possède une valeur: celle de ***i*** après affectation
- Cet opérateur peut faire intervenir d'autres expressions comme: ***c = b + 3***
- La priorité de = est inférieure à celle de tous les opérateurs arithmétiques et de comparaison: <u>on évalue ce qui est à droite et on l'affecte</u> à la "valeur à gauche" ou *"left value"* ou ***lvalue***

➢ **<u>Rappel:</u>**

c + 5 = x *n'a pas de sens* **!**

Opérateurs et expressions

Opérateur d'affectation simple:

<u>Associativité:</u> ***i = j = 5***

On évalue la droite **j = 5** avant d'en affecter la valeur à la *lvalue* ***i***. C'est l'associativité à gauche.

<u>Conversions liées aux affectations:</u>

Contrairement aux autres opérateurs il n'est plus question d'effectuer de conversion de la *lvalue*:

➢ Si le type de l'expression figurant à droite n'est pas du même type que la *lvalue* il y a conversion systématique dans le type de la *lvalue.* Ceci peut conduire à une **dégradation de l'information**

Opérateurs et expressions

Opérateurs d'incrémentation et de décrémentation:

On utilise souvent des expressions ***i = i +1*** ou ***n = n – 1***

En C ces actions peuvent être réalisées par des opérateurs unaires: ***++i i++ --i i --***

L'expression ***++i*** a pour effet d'incrémenter de **1** la valeur de ***i*** et sa valeur est celle de ***i*** après incrémentation.

On dit que ++ est un opérateur de:

pré incrémentation lorsqu'il est **à gauche** de la *lvalue*
post incrémentation lorsqu'il est **à droite** de la *lvalue*

De même -- est un opérateur de:

pré décrémentation lorsqu'il est **à gauche** de la *lvalue*
post décrémentation lorsqu'il est **à droite** de la *lvalue*

Opérateurs et expressions

Opérateurs d'incrémentation et de décrémentation:

<u>Priorité:</u>

Les priorités élevées de ces opérateurs unaires permettent d'écrire des expressions compliquées sans parenthèses.
Exemple:

3 * i++ * j-- + k++ a un sens!

<u>Intérêt:</u>

- Allègent l'écriture des expressions
- On ne cite qu'une fois la *lvalue* (moins de risque d'erreur)

Opérateurs et expressions

Opérateurs d'affectation élargie:

On dispose d'opérateurs encore plus puissant:

On peut remplacer ***i = i +k*** par ***i + = k***
ou ***a = a * b*** par ***a * = b***

D'une manière générale on peut condenser les affectations de la forme:

lvalue = lvalue opérateur expression
en
lvalue opérateur = expression

pour tous les opérateurs binaires arithmétiques et de manipulation de bits

Opérateurs et expressions

Opérateurs de forçage de type ou **opérateur de cast:**

On peut forcer la conversion d'une expression dans un type de son choix:

Si ***n*** et ***p*** sont des entiers, ***(double) (n/p)*** convertie d'abord n en double et l'opération sera alors évalué en double.

➢ C'est un opérateur unaire qui fait la conversion en double du résultat de l'expression mais pas des différentes valeurs.

➢ La priorité de ces opérateurs est élevée.

➢ Il existe autant d'opérateurs de ***cast*** que de types différents

Opérateurs et expressions

Opérateur conditionnel:

Considérons la structure de choix:

Si a>b alors max=a sinon max=b

Il existe un opérateur qui réalise la même tâche:

max = a > b ? a : b

L'expression à droite de l'opérateur d'affectation est en fait constitué de **3 opérandes** séparés par les symboles ***?*** et ***:***

Cet opérateur évalue la première expression, si elle est différente de zéro, il évalue le deuxième opérande et donne le résultat sinon il évalue le troisième opérande et fourni le résultat

Sa priorité est faible; juste avant l'affectation.

Opérateurs et expressions

Opérateur séquentiel:

La notion d'expression est très large en C.

a * b , i + j

est une expression qui évalue d'abord ***a * b*** puis ***i + j*** et qui prend la valeur de ***i + j***

- Dans cet exemple le premier calcul est inutile, par contre

i ++ , a + b

peut représenter un intérêt …

- L'opérateur séquentiel **","** est associatif de gauche à droite, sa faible priorité évite l'usage des parenthèses:

i ++ , j = i + k , j --

- Il permet de réunir plusieurs instructions en une seule.
- Dans la pratique il est fréquemment utilisé dans les boucles et les instruction de choix

Opérateurs et expressions

Opérateurs de manipulation de bits:

C'es opérateurs permettent de travailler directement sur le "motif binaire", ils ne portent que sur les types entiers

Type	Opérateur	Signification
Binaire	&	Et (bit à bit)
Binaire	|	Ou inclusif (bit à bit)
Binaire	^	Ou exclusif (bit à bit)
Binaire	<<	Décalage à gauche
Binaire	>>	Décalage à droite
Unaire	~	Complément à un (bit à bit)

Opérateurs et expressions

Opérateurs de manipulation de bits: Exemples

Variable / Opérateur	En binaire	Hexa.	Décimal
n	0000 0101 0110 1110	056E	1390
p	0000 0011 1011 0011	03B3	947
n & p	0000 0001 0010 0010	0122	290
n | p	0000 0111 1111 1111	07FF	2047
n ^ p	0000 0110 1101 1101	06DD	1757
~ n	1111 1010 1001 0001	FA91	-1391(signed)
~ n	1111 1010 1001 0001	FA91	64145 (unsigned)

Opérateurs et expressions

Opérateurs de manipulation de bits: Exemples

Variable / Opérateur	Cas 1	Cas 2
(signed) n	0000 0101 0110 1110	1010 1101 1101 1110
(unsigned) p	0000 0101 0110 1110	1010 1101 1101 1110
n << 2	0001 0101 1011 1000	1011 0111 0111 1000
n >> 3	0000 0000 1010 1101	1111 0101 1011 1011
p >> 3	0000 0000 1010 1101	0001 0101 1011 1011

n << 2: décale n de 2 bits vers la gauche: 2 bits de poids fort sont perdus et 2 bits à 0 apparaissent à droite.

n >> 3: décale n de 3 bits vers la droite: 3 bits de droite sont perdus et 3 bits apparaissent à gauche.

- si n est ***unsigned*** les bits créés sont à 0.
- si n est ***signed*** les bits ajoutés sont identiques au bit de signe

Opérateurs et expressions

Opérateur *sizeof*:

- Il retourne la taille d'un objet en octets. Dans ce cas on ne met pas de parenthèses
- Il donne la taille en nombre d'octets du type dont le nom est entre parenthèses

double n;
unsigned int size_of_an_int = sizeof (int);
unsigned int size_of_n = sizeof n;

- **Intérêt**:
  - Ecrire des programmes portables en connaissant la taille exacte de certains objets
  - Eviter de calculer la taille des objet par soi-même d'un type complexe

Priorités des opérateurs

Rang	Opérateur	ordre de priorité
1	()	gauche à droite
2	~ ++ -- (type)	droite à gauche
3	* / %	gauche à droite
4	+ -	gauche à droite
5	<< >>	gauche à droite
6	< <= > >=	gauche à droite
7	== !=	gauche à droite
8	&	gauche à droite
9	^	gauche à droite
10	|	gauche à droite
11	&&	gauche à droite
12	||	gauche à droite
13	Op. conditionnel	droite à gauche
14	=	droite à gauche

Les instructions de contrôle

La puissance d'un langage provient de la possibilité de faire:

- Des comportement différent selon les circonstances
- De répéter des instructions
- Des branchements conditionnels ou inconditionnels

En C ces instructions de contrôle sont:

if … else et ***switch***	pour les choix
do … while, ***while*** et ***for***	pour les boucles
goto, ***break*** et ***continue***	pour les branchements

Les instructions de contrôle

L'instruction *if*:

if (expression)
instruction_1
else
instruction_2

else et l'instruction qui le suit sont facultatifs. On peut avoir:

if (expression)
instruction

Avec:

expression:	expression quelconque
Instruction:	instructions simple, bloc d'instructions, instruction structurée

Les instructions de contrôle

L'instruction *if*: imbrications

if (a<b)
if (b<c)
printf ("ordonné");
else
printf ("non ordonné");

- Il y a une ambiguïté sur le ***else***

La regle est:

***else* se rapporte au dernier if rencontré auquel un *else* n'a pas été attribué**.

- L'introduction des accolades outrepasse cette règle!

Les instructions de contrôle

L'instruction *switch*:

switch (expression)
{ case constante_1 : [instruction_1]
case constante_2 : [instruction_2]
…………….
case constante_n : [instruction_n]
[default : instructions]
}

expression: expression **entière** quelconque
constante: expression **constante** de type entier
instruction: instructions quelconques

- Les crochets **[ ]** indiquent que le terme est facultatif

Les instructions de contrôle

L'instruction *switch*:

switch (expression)
{ case constante_1 : [instruction_1]
case constante_2 : [instruction_2]
…………….
case constante_n : [instruction_n]
[default : instructions]
}

- On évalue l'expression puis on cherche s'il existe une étiquette avec à la valeur obtenue. Si c'est le cas on exécute les instructions figurant après cette étiquette. Sinon si le mot clé ***default*** existe on exécute les instructions qui le suivent, sinon on passe à l'instruction qui suit le bloc.
- L'instruction ***break*** faire sortir du bloc ***switch***

Les instructions de contrôle

L'instruction *switch*:

switch (expression)
{ case constante_1 : [instruction_1]
case constante_2 : [instruction_2]
…………….
case constante_n : [instruction_n]
[default : instructions]
}

- Une fois le branchement à l'étiquette de cas effectué, l'exécution se poursuit, par défaut, jusqu'à la fin du bloc ***switch***
- L'instruction d'échappement ***break*** permet de forcer la sortie du bloc

Les instructions de contrôle

L'instruction *do…while*:

do instruction
while (expression) ;

Action: répète **instruction** tant que **expression** est vraie.

- L'instruction ou bloc d'instructions est parcourue **au moins une fois** parce que la condition qui régie cette boucle (valeur de expression) n'est examinée qu'a la fin de chaque répétition.
- Notez la présence de la parenthèse autour de l'expression de contrôle et le ***;*** à la fin de cette instruction
- L'expression peut être aussi élaborée que vous voulez
- L'instruction peut être vide.

Exemple: ***do ; while (...)*** ou ***do { } while (…)***

Les instructions de contrôle

L'instruction *do…while*:

Exemple 1:

do { printf ("donnez un nb > 0 : ") ;
scanf ("%d" , &n) ;
printf (" vous avez fourni %d ", n);
} while (n <= 0) ;

Exemple 2

do c = getchar();
while (c != 'x') ;

Les instructions de contrôle

L'instruction *while*:

while (expression)
instruction

Action: répète **instruction** tant que **expression** est vraie.

- La condition de poursuite est examinée avant chaque exécution de l'instruction ou bloc d'instructions. Une telle boucle peut très bien n'être parcourue aucune fois si la condition est fausse dès le départ.
- Notez la présence de la parenthèse autour de l'expression de contrôle et l'absence du ; (il y a déjà un dans instruction !)
- L'expression est évaluée avant la boucle. Sa valeur doit être définie …

Les instructions de contrôle

L'instruction *while*:

Exemple:

int k=0, nbr=23;
while (nbr!=0)
{ nbr /= 2; k ++;
}
printf("%d\n",k);

Les instructions de contrôle

L'instruction for:

for ([expression_1]; [expression_2]; [expression_3])
instruction

Action: Répète **instruction** tant que **expression_2** est vraie.

- Les crochets [ ] signifient que leur contenu est facultatif
- Elle est équivalente à:

expression_1
while (expression_2)
{ instruction
expression_3
}

- Si **expression_2** est vide elle est considérée comme vraie

Les instructions de contrôle

L'instruction *for*:

Exemple:

```
int i ;
for (i=0 ; i<4 ; i++ )
{
    printf(%d\n",i*i);
}
printf("a la fin la valeur de i est %d\n",i);
```


Les entrées-sorties conversationnelles

La fonction *printf*

printf est utilisée pour transférer du texte, des valeurs de variables ou des résultats d'expressions vers le fichier de sortie standard ***stdout*** (par défaut l'écran).

Syntaxe: ***printf (format, liste_d'expressions)***

Avec:

format: format de représentation est en fait une chaîne de caractères qui peut contenir:

- du texte
- des séquences d'échappement
- des spécificateurs de format

liste_d'expressions: suite d'expressions séparées par des virgules d'un type en accord avec le format correspondant **"<format>"**

Les entrées-sorties conversationnelles

La fonction *printf*

- Les **séquences d'échappement** sont des couples de symboles qui contrôlent l'affichage ou l'impression de texte.
- Ces séquences d'échappement sont toujours précédées par le caractère d'échappement '\'

\a	sonnerie	\\	trait oblique
\b	curseur arrière	\?	point d'interrogation
\t	tabulation	\'	apostrophe
\n	nouvelle ligne	\"	guillemets
\r	retour au début de ligne	\f	saut de page (imprimante)
\0	NUL	\v	tabulateur vertical

Les entrées-sorties conversationnelles

La fonction *printf*

- Les **spécificateurs de format** indiquent la manière dont les valeurs des expressions sont imprimées. Ils commencent toujours par le symbole **%** et se terminent par un ou deux caractères qui indiquent le format d'impression.
- Les spécificateurs de format impliquent une **conversion** d'un nombre en chaîne de caractères.
- La partie **"<format>"** contient exactement un spécificateur de format pour chaque expression.

Les entrées-sorties conversationnelles

La fonction *printf*

Spécificateurs de format pour: *printf*

Symbole	Type	Impression comme
%d* ou *%i	***int***	entier relatif
%u	***int***	entier naturel (unsigned)
%o	***int***	entier exprimé en octal
%x	***int***	entier exprimé en hexadécimal
%c	***int* ou *char***	caractère

Pour le type ***long***, il faut utiliser les spécificateurs:

%ld %li %lu %lo %lx

Les entrées-sorties conversationnelles

La fonction *printf*

Spécificateurs de format pour: *printf (suite)*

Symbole	Type	Impression comme
%f	***double***	rationnel en notation décimale
%e	***double***	rationnel en notation scientifique
%s	***char****	chaîne de caractères

Pour le type ***long double***, il faut utiliser les spécificateurs

%Lf ou ***%Le***

Les entrées-sorties conversationnelles

La fonction *printf*

Longueur minimale de cadrage:

```
printf("%3d",n);
        n= 20        ^20
        n= 3         ^^3
        n=2358       2358
        n=-5200      -5200
printf("%-3d",n);
        n= 20        20^
        n= 3         3^^
        n=2358       2358
```

- Le code ***%3d*** affiche une valeur de type ***int*** sur **au moins trois caractères** cadrés à droite
- Le code ***%-3d*** affiche une valeur de type ***int*** sur au moins trois caractères mais cadrés à gauche

Les entrées-sorties conversationnelles

La fonction *printf*

```
printf("%f",x);
        x= 1.2345           1.2345000
        x= 12.3456789       12.345679
printf("%10f",x);
        x= 1.2345           ^^1.234500
        x= 12.345           ^12.345000
        x= 1.2345E5         123450.000000
printf("%10.3f",x);
        x= 1.2345           ^^^^^1.235
        x= 1.2345E3         ^^1234.5500
        x= 1.2345E7         12345000.000
```

- ***%f*** affiche la valeur avec 6 chiffres après la virgule.
- ***%10f*** précise une longueur minimale de 10 caractères
- ***%10.3f*** précise une longueur minimal de 10 avec 3 chiffres après la virgule

Les entrées-sorties conversationnelles

La fonction *printf*

```
printf("%e",x);
        x= 1.2345           1.2345e+000
        x= 123.45           1.234500e+002
        x= 123.456789E8     1.234568e+010
        x= -123.456789E8    -1.234568e+010

printf("%12.4e",x);
        x= 1.2345           ^1.2345e+000
        x= 23.456789E8      ^1.2346e+010
```

- ***%e*** affiche en notation exponentielle sur 14 caractères dont 3 pour l'exposant et max 6 après la virgule
- ***%12.4e*** précise une longueur 12 et une précision 4. ici la valeur affichée occupera 12 caractères

Les entrées-sorties conversationnelles
La fonction *printf*

La valeur de retour de printf:

```
p = printf("%3d",n);
        n= 25            on affiche  ^25      et    p vaudra 3
        n= -4583         on affiche  -4583    et    p vaudra 5
```

➢ ***printf*** étant une fonction elle retourne le nombre de caractères qu'elle a effectivement affiché ou la valeur -1 en cas d'erreur.

Ceci peut se révéler intéressant si on veut contrôler la mise en page et ne forcer le retour à la ligne que quand c'est nécessaire.

Les entrées-sorties conversationnelles
La fonction *printf*

C cherche toujours à satisfaire le contenu du format !

printf("%d", n, p) → la valeur de p ne sera pas affichée

printf("%d %d", n) → affichera n puis … n'importe quoi.

Les entrées-sorties conversationnelles
La macro *putchar*

Nous avons déjà employé ***putchar*** et nous pouvons dire que ***putchar(c)*** joue le même rôle que ***printf("%c",c)***

Mais son exécution est plus rapide dans la mesure où elle ne fait pas appel au mécanisme d'analyse de format.

Les entrées-sorties conversationnelles
La fonction *scanf*

scanf est la fonction symétrique à ***printf*** qui reçoit ses données à partir du fichier d'entrée standard ***stdin*** (clavier par défaut)

Syntaxe: ***scanf (format, liste_d'adresses)***

Avec:
format: des spécificateurs de format (entre " ") qui détermine comment les données reçues doivent être interprétées.
liste_d'adresses: liste d'adresses de "***lvalue***" séparées par des virgules d'un type en accord avec le format correspondant

- Les données correctement reçues sont mémorisées successivement aux adresses indiquées par ***liste_d'adresses***.
- L'adresse d'une variable est indiquée par le nom de la variable précédé du signe ***&***

Les entrées-sorties conversationnelles
La fonction *scanf*

Rôle des séparateurs:

scanf ("%d", &n)

```
12 ⏎            n=12
^12 ⏎           n=12
^12^^ ⏎         n=12
⏎
⏎
^12^^ ⏎         n=12
```

scanf ("%d^%d", &n, &p) ou scanf ("%d%d", &n, &p)

```
12^25⏎          n=12   p=25
12 ⏎
25 ⏎            n=12   p=25
```


Les entrées-sorties conversationnelles
La fonction *scanf*

Quand on impose une longueur maximale:

scanf("%3d^%3d", &n, &p) ou scanf("%3d%3d", &n, &p)

```
123456⏎             n=123  p= 456
^^123456⏎           n=123  p= 456
^^123^^^45⏎         n=123  p= 45
```

Lecture de caractères:

scanf ("%d^%c", &n, &c)

```
12^a⏎         n=12    c = 'a'
12^^^a⏎       n=12    c = 'a'
12⏎ a⏎        n=12    c = 'a'
12a⏎          n=12    c = 'a'
```

scanf ("%d%c", &n, &c)

```
12a⏎          n=12    c = 'a'
12^a⏎         n=12    c = '^'
```


Les entrées-sorties conversationnelles
getchar()

getchar, lit un caractère du fichier d'entrée standard ***stdin***.

Type du résultat

le type résultat de ***getchar*** est ***int***

Les valeurs retournées sont ou bien des caractères (0 - 255) ou bien le symbole EOF.

En général, ***getchar*** est utilisé dans une affectation:

```
int C;
C = getchar();
```


Les types composés

A partir des types prédéfinis du C (char, entiers, flottants), on peut créer de nouveaux types qui permettent de représenter des ensembles de données organisées.

- **Les tableaux**: ensemble fini d'éléments de même type, stockés en mémoire à des adresses contiguës
- **Les structures**: suite finie d'objets de types différents appelé champ et désigné par un identificateur
- Les champs de bits
- Les unions
- Les énumérations

Les types composés

Les tableaux unidimensionnels: ensemble fini d'éléments de même type, stockés en mémoire à des adresses contiguës, référencées par un indice

Correspond en mathématiques à un vecteur de dimension N

A: [| | | . . . | |]
N composantes

Déclaration d'un tableau à une dimension:

type nom-du-tableau[nombre-éléments];

nombre-éléments est une expression constante entière positive.

int tab[10];

indique que ***tab*** est un tableau de ***10*** éléments de type ***int***.

Cette déclaration réserve pour ***tab*** un espace mémoire consécutifs de 10 cases de type ***int***

Les types composés

Les tableaux unidimensionnels :

- les indices d'un tableau ***t*** de taille ***n*** vont de **0** à **n-1**.
  Tout accès à ***t[n]*** peut provoquer une erreur grave pendant l'exécution du programme.
- un tableau doit avoir une taille fixe connue à la compilation.
  Cette taille peut être un nombre ou une variable constante, mais **pas une variable**.

Les types composés

Initialisation de tableaux unidimensionnels :

Elle peut se faire au moment de la déclaration:

int tab[6] = {1,3,8,0,5,9};

Si l'initialisation et la déclaration sont simultanées, la taille du tableau peut être omise, le compilateur la calcule d'après le nombre de valeurs d'initialisation:

int tab[] = {10,20,30,40,50,60,70,80,90};

On peut n'initialiser que le début d'un tableau:

double resistances[12]={1., 1.2, 1.8, 2.2};

- Si on spécifie trop de valeurs, un message d'erreur le signale.
- Si tout le tableau n'est pas initialisé lors de la déclaration, les éléments non initialisés sont mis à 0.

Les types composés

Initialisation de tableaux unidimensionnels:

Un tableau peut être initialisé plus tard, en affectant une valeur à chaque élément :

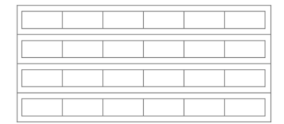
```
int notes[56], i;
for (i=0; i<56; i++)
{
notes[i]=20;
}
notes[3] = 13;
notes[55] = 9;
```

- Il n'y a pas d'affectation, de comparaison, d'opérations arithmétiques directes sur les tableaux.

Les types composés

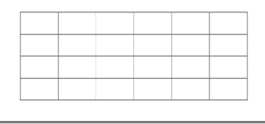
Les tableaux multidimensionnels: Dimension 2

Un tableau à deux dimensions L et C est un tableau de dimension L dont chaque élément est un tableau de C éléments.



L est le nombre de lignes, C est le nombre de colonnes

Un tableau à deux dimensions contient donc L*C composantes.



Les types composés

Les tableaux multidimensionnels: Dimension 2

Un tableau à deux dimensions L et C correspond en mathématique à une matrice de L lignes et C colonnes.

Déclaration d'un tableau à une dimension:

type nom-du-tableau[nb-lignes] [nb-colonnes];

Exemple ***int tab[4][6];***

indique que ***tab*** est un tableau de ***4*** lignes et 6 colonnes tous de type ***int***.

Cette déclaration réserve pour ***tab*** un espace mémoire consécutifs de 4*6=24 cases de type ***int***

Chaque élément est identifié par 2 entiers: l'indice de ligne et celui de la colonne

Les types composés

Les tableaux multidimensionnels: Dimension 2

Exemple tableau de dimension 2:

```
main()
{
 int tab[5][10];
 int i,j;
 { … }        /* affectations…*/
 for (i=0; i<5; i++)    /* Pour chaque ligne ... */
    {
      for (j=0; j<10; j++) /* pour chaque colonne */
        printf("%7d", tab[i][j]);
      printf("\n");   /* Retour à la ligne */
    }
}
```


Les types composés

Les tableaux multidimensionnels: Dimension 2

On peut initialiser les composantes du tableau lors de sa déclaration, en donnant la liste des valeurs entre accolades.

les composantes de chaque ligne sont encore une fois comprises entre accolades.

```
int A[3][10] ={{ 0,10,20,30,40,50,60,70,80,90},
               {10,11,12,13,14,15,16,17,18,19},
               { 1,12,23,34,45,56,67,78,89,90}};
float B[3][2] = {{-1.05,   -1.10  },
                 {86e-5,    87e-5 },
                 {-12.5E4, -12.3E4}};
```

Les valeurs sont affectées ligne par ligne en passant de gauche à droite. Les valeurs manquantes sont initialisées par zéro.

Les types composés

Les tableaux multidimensionnels: Dimension 2

Exemples d'initialisations

```
int C[4][4] = {{1, 0, 0, 0}
               {1, 1, 0, 0}
               {1, 1, 1, 0}
               {1, 1, 1, 1}};
```

C:

1	0	0	0
1	1	0	0
1	1	1	0
1	1	1	1

```
int C[4][4] ={{1, 1, 1, 1}};
```

C:

1	1	1	1
0	0	0	0
0	0	0	0
0	0	0	0

```
int C[4][4] ={{1}, {1}, {1}, {1}};
```

C:

1	0	0	0
1	0	0	0
1	0	0	0
1	0	0	0

```
int C[4][4] ={{1, 1, 1, 1, 1}};
```

ERREUR !

Les types composés

Les tableaux multidimensionnels: Dimension 2

Réservation automatique

Si le nombre de lignes n'est pas indiqué lors de l'initialisation, l'ordinateur réserve automatiquement le nombre d'octets nécessaires.

```
int A[][10] = {{ 0,10,20,30,40,50,60,70,80,90},
               {10,11,12,13,14,15,16,17,18,19},
               { 1,12,23,34,45,56,67,78,89,90}};
```

réservation de 3*10

```
float B[][2] = {{-1.05,   -1.10  },
                {86e-5,    87e-5 },
                {-12.5E4, -12.3E4}};
```

réservation de 3*2

Les types composés

Les tableaux multidimensionnels: Cas général

Il se définit par:

type Nom_du_tableau [a1][a2][a3] ... [aN]

Chaque élément entre crochets désigne le nombre d'éléments dans chaque dimension

Le nombre de dimensions n'est pas limité

Un tableau à N dimensions est un tableau de tableaux à N-1 dimensions.

Les types composés
Cas particulier des chaines de caractères:

Une chaîne de caractères **est un tableau** qui ne contient que des éléments de type ***char*** et terminé par le caractère ***'\0'*** .

char mot[6] = "F.S.R"; correspond au tableau :

[0]	[1]	[2]	[3]	[4]	[5]
F	**.**	**S**	**.**	**R**	**'\0'**

Et ***char ville[10] = "Rabat";***

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
R	**a**	**b**	**a**	**t**	**'\0'**				

On peut omettre la taille du tableau lors de la déclaration, s'il y a initialisation: ***char ville[] = "Rabat";***

[0]	[1]	[2]	[3]	[4]	[5]
R	**a**	**b**	**a**	**t**	**'\0'**

L'accès à un élément d'une chaîne de caractères se fait de la même façon que l'accès à un élément d'un tableau.

Les types composés
Cas particulier des chaines de caractères:

Les bibliothèques de fonctions de C contiennent une série de fonctions spéciales pour le traitement de chaînes de caractères.

Exemple: ***string.h***

strlen(s) fournit la longueur de ***s*** sans compter le ***'\0'***

strcpy(s,t) copie ***t*** vers ***s***

strcat(s, t) ajoute ***t*** à la fin de ***s***

strcmp(s, t) compare ***s*** et ***t*** lexicographiquement et fournit un résultat:
- négatif si ***s*** précède ***t***
- zéro si ***s*** est égal à ***t***
- positif si ***s*** suit ***t***

strncpy(s, t, n) copie au plus n caractères de ***t*** vers ***s***

strncat(s, t, n) ajoute au plus n caractères de ***t*** à la fin de ***s***

Les types composés
Cas particulier des chaines de caractères:

stdlib.h contient des déclarations de fonctions pour la conversion de chaînes de caractères en nombres

atoi(s) retourne la valeur numérique représentée par ***s*** comme ***int***

atol(s) retourne la valeur numérique représentée par ***s*** comme ***long***

atof(s) retourne la valeur numérique représentée par ***s*** comme ***double***

- Les espaces au début d'une chaîne sont ignorés
- Il n'y a pas de contrôle du domaine de la cible
- La conversion s'arrête au premier caractère non convertible
- Pour une chaîne non convertible, ces fonctions retournent zéro

Les types composés
Cas particulier des chaines de caractères:

Fonctions de classification et de conversion ***ctype.h***

fonction:	**retourne un *int* différent de zéro,**
isupper(c)	si ***c*** est une majuscule ('A'...'Z')
islower(c)	si ***c*** est une minuscule ('a'...'z')
isdigit(c)	si ***c*** est un chiffre décimal ('0'...'9')
isalpha(c)	si ***islower(c)*** ou ***isupper(c)***
isalnum(c)	si ***isalpha(c)*** ou ***isdigit(c)***
isxdigit(c)	si ***c*** est un chiffre hexadécimal
isspace(c)	si ***c*** est un signe d'espacement (' ','\t','\n','\r','\f')
tolower(c)	retourne ***c*** converti en minuscule
toupper(c)	retourne ***c*** converti en majuscule

Les types composés
Cas particulier des chaines de caractères:

Pour mémoriser une suite de mots, il est pratique de considérer un tableau de chaînes de caractères.

Il correspond à un tableau à deux dimensions du type ***char***, où chaque ligne contient une chaîne de caractères:

char jour[7][9]; réserve l'espace en mémoire pour 7 mots contenant 9 caractères (dont 8 caractères significatifs)

On accède aux différentes chaînes de caractères du tableau, en indiquant simplement la ligne correspondante.

```
char jour[7][9]= {"lundi", "mardi", "mercredi", "jeudi",
        "vendredi", "samedi", "dimanche"};
printf("Aujourd'hui, c'est %s !\n", jour[2]);
```

affichera la phrase:

Aujourd'hui, c'est mercredi !

Les types composés

Le langage C permet de créer de nouveaux types de variables en utilisant la notion de **structure.**

C'est une suite finie d'objets de types différents appelés champs et désigné par un seul identifiant.

La déclaration d'une structure ne fait que donner l'allure de la structure:

```
struct modele
{
    type_1 membre_2;
    type_2 membre_2;
    …
    type_n membre_n;
};
```

Elle ne réserve pas d'espace mémoire.

Les types composés

La définition d'une variable structurée consiste à créer une variable ayant comme type celui d'une structure que l'on a précédemment déclarée

Pour une variable d'identifiant ***X*** de type ***modele*** on utilise la syntaxe: ***struct modele X;***

On peut aussi faire les deux déclarations (du type et de la variable) en même temps:

```
struct modele
{
    type_1 membre_1;
    type_2 membre_2;
    …
    type_n membre_n;
} X;
```


Les types composés

Exemple: dans un plan un point est identifié par ses coordonnées x et y. on crée une structure nommé ***coordonnées*** .

```
struct coordonnées
{
    int x; // Abscisses
    int y; // Ordonnées
};
```

Si on a besoin de 3 points_a, point_b et point_c on déclare:

struct coordonnées points_a, point_b, point_c;

Chacun de ces points à 2 coordonnées x et y;

Pour y accéder, il suffit de mettre un point entre le nom de la variable et le champs désiré:

```
point_b.x=3; points_b.y=12;
points_a.x= points_b.y;
points_a.y= points_b.x;
```

Les types composés

Utilisation de *typedef*

L'instruction ***typedef*** crée un alias

typedef struct coordonnees coordonnees;

Cette ligne indique:

"le mot ***coordonnees*** est équivalent à ***struct coordonnees***"

En mettant cette instruction, on n'aura plus besoin de mettre le mot ***struct*** à chaque définition

Exemple:

coordonnees a, b;

déclare 2 variables nommées a et b de type ***struct coordonnees***

Les types composés

Les structures se manipulent comme les autres types. La définition, l'affectation, l'initialisation, … tout est semblable au comportement des types de base.

On utilise des accolades pour préciser les valeurs des champs en cas d'initialisation (comme pour un tableau).

On peut évidemment déclarer des tableaux de structures et même définir un champ de type structure ou tableau …

coordonnees a={2,13},b=a, c, tab[10];
c=a;

- Par contre écrire ***b={0,2};*** donne **une erreur!**
- Comme pour les tableaux, la syntaxe utilisée pour l'initialisation ne marche pas pour une affectation.

Les types composés

Exemple:

```
struct Personne
{   char nom[100];
    char prenom[100];
    char adresse[1000];
    int age;
    int garcon; };
typedef struct Personne fiche;
main()
{   fiche utilisateur;
    printf("Quel est votre nom ? ");
    scanf("%s", utilisateur.nom);
    printf("Votre prenom ? ");
    scanf("%s", utilisateur.prenom);
    printf("Vous vous appelez %s %s", utilisateur.prenom,
   utilisateur.nom);
}
```

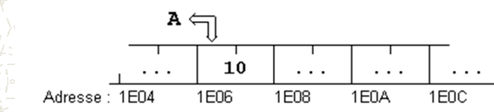

Les pointeurs

- La mémoire d'une machine est constituée de cases (généralement un octet) appelées blocs.
- Chacun de ces blocs est identifié par un numéro. C'est l'adresse du bloc.
- Une variable, selon son type, occupe un ou plusieurs blocs consécutifs.
- l'adresse d'une variable correspond à l'adresse du premier bloc réservé à la variable.
- l'adresse d'une variable change à chaque exécution.

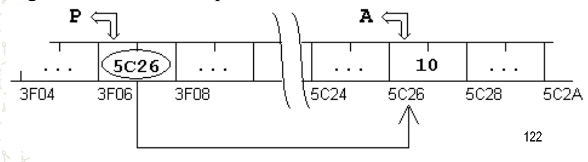
Les pointeurs

On peut accéder au contenu d'une variable de 2 façons :

- grâce à son identifiants (accès direct)



- grâce à l'adresse du premier bloc alloué à la variable



Les pointeurs

Un pointeur est une variable qui peut contenir l'adresse d'une autre variable.

La valeur d'un pointeur est toujours un entier (éventuellement long).

Un pointeur est limité à un type de données: Il peut contenir l'adresse d'une variable de ce type ou l'adresse d'un élément d'un tableau de ce type.

Il faut bien faire la différence:

- Un pointeur est une variable qui peut 'pointer' sur différentes adresses.
- Le nom d'une variable reste toujours lié à la même adresse.

Les pointeurs

Pour travailler avec des pointeurs nous avons besoin de:

- Déclarer un pointeur
- L'opérateur 'adresse de': & pour obtenir l'adresse d'une variable.
- L'opérateur 'contenu de': * pour accéder au contenu d'une adresse.

L'opérateur & ne s'applique qu'aux variables et les tableaux. Il ne peut pas être appliqué aux constantes ou aux expressions.

Si P est un pointeur et A une variable (du même type) contenant la valeur 10. L'instruction P = &A; affecte l'adresse de la variable A à la variable P.

Les pointeurs

Déclaration d'un pointeur:

Un pointeur est une variable qui doit être définie en précisant le type de variable pointée. La syntaxe est:

type * Nom_du_pointeur

Le type de variable pointée peut être un type primaire (char, int, float, ...) ou un type complexe (struct...).

Le symbole '*' indique qu'il s'agit d'une variable de type pointeur

Le type indique au compilateur la taille des zones pointées.

Les pointeurs

Opération sur les pointeurs:

Après l'instruction ***P = &X;*** les expressions suivantes, sont équivalentes:

*Y = *P+1*	*Y = X+1*
**P = *P+10*	*X = X+10*
**P += 2*	*X += 2*
*++*P*	*++X*
*(*P)++*	*X++*

- On ne peut affecter que des adresses à un pointeur. Seule exception: La valeur 0 (***NULL***) pour indiquer qu'un pointeur ne pointe 'nulle part'.

Les opérateurs ******* et ***&*** ont la même priorité que les opérateurs unitaires (***!*** , ***++*** , ***--***).

Les pointeurs

Opération sur les pointeurs:

Les pointeurs sont des variables et peuvent être utilisés comme telles.

Si P1 et P2 sont deux pointeurs sur ***int***, alors P1 = P2; copie le contenu de P2 vers P1. (P1 pointe alors sur le même objet que P2)

Après les instructions:

int A, *P; P = &A;

A	désigne le contenu de A
&A	désigne l'adresse de A
P	désigne l'adresse de A
****P***	désigne le contenu de A
&P	désigne l'adresse du pointeur P
****A***	est illégal (puisque A n'est pas un pointeur)

Les pointeurs

Adressage des composantes d'un tableau:

En déclarant un tableau A de type ***int*** et un pointeur P sur ***int***,
int A[10], *P;

l'instruction: ***P = A;*** est équivalente à ***P = &A[0];***

Si P pointe sur une composante quelconque d'un tableau, alors P+1 pointe sur la composante suivante:

P+i	pointe sur la i-ième composante derrière P
P-i	pointe sur la i-ième composante devant P.

Ainsi, après l'instruction ***P = A;***

le pointeur P pointe sur A[0]

*(P+1)	désigne le contenu de A[1]
*(P+2)	désigne le contenu de A[2]
…	…
*(P+i)	désigne le contenu de A[i]

Les pointeurs

Adressage des composantes d'un tableau:

Soustraction de deux pointeurs:

- Si P1 et P2 deux pointeurs qui pointent dans le même tableau: P1-P2 fournit le nombre de composantes comprises entre P1 et P2.
- Le résultat de P1-P2 est négatif, si P1 précède P2
  zéro, si P1 = P2
  positif, si P2 précède P1
- Le résultat de P1-P2 est indéfini, si P1 et P2 ne pointent pas dans le même tableau
- La comparaison de 2 pointeurs qui pointent dans le même tableau est équivalente à la comparaison des indices correspondants.

(Si les pointeurs ne pointent pas dans le même tableau, le résultat est donné par leurs positions relatives dans la mémoire).

Les fonctions

Un programme devient difficile a comprendre dés qu'il dépasse une ou deux pages:

- L'écriture modulaire permet de diviser en plusieurs parties le programme et de regrouper dans le "programme principal" les enchaînements de ces sous programmes.
- Chacun de sous programmes peut être décomposée en modules plus élémentaires

La "programmation structurée":

- Permet d'éviter des séquences d'instructions répétitives.
- Permet le partage des modules qu'il suffit d'avoir écrits et mis au point une seule fois

Les fonctions

En C il existe un seul type de module: La FONCTION.

La fonction reçoit des arguments et fournit UN résultat

Une fonction est constituée de:

- entête: type et identifiant de la fonction suivis d'une liste d'arguments entre parenthèses
- instruction composée constituant le corps de la fonction

la fonction doit être déclarée avant que le compilateur rencontre le premier appel:

- Au début de la fonction main()
- Avant la définition de la fonction main()
- Dans un fichier séparé (.h) (qu'il faut inclure avec la directive #include !)

Les fonctions

```
main()
{
float fexple (float, int, int);      /* déclaration de la fonction */
float x = 1.5, float y, z;
int n = 3, p= 5, q= 10;
y = fexple (x, n, p);                /* appel de fexple avec x, n et p */
printf ("valeur de y : %e\n", y);
z = fexple (x+0.5, q, n-1);          /*appel de fexple avec formules*/
printf ("valeur de z : %e\n", z) ;
}
/**********Définition de la fonction fexple ************/
float fexple (float x, int b, int c)
{  float val;    /* déclaration d'une variable "locale" a fexple*/
val = x * x + b * x + c ; return val;
}
```


Les fonctions

❑ Arguments muets et arguments effectifs

- Les arguments figurant dans la définition de la fonction sont des arguments muets. Ils permettent de décrire ce qu'elle doit faire.
- Les arguments fournis lors de 1'appel sont des arguments effectifs. On peut utiliser n' importe quelle expression comme argument effectif.

❑ L'instruction return

- Peut mentionner toute expression. Ex: *return(x*x+b*x+c);*
- Peut apparaître à plusieurs reprise dans une fonction
- Elle fixe la valeur du résultat **ET** arrête l'exécution de la fonction

Les fonctions

Quelques règles:

Une fonction peut ne pas retourner de valeurs:

void sansval (int n)

dans ce cas pas de *return* !

Une fonction peut ne pas avoir d'arguments:

float tirage (void)

Une fonction peut ne pas retourner de valeurs et ne pas avoir d'arguments:

void message (void)

Les fonctions

```
main()
{
void affiche-carres (int, int) ;
void erreur (void) ;
int debut = 5, fin = 10 ;
  ...
affiche-carres (debut, fin);
  ...
if (...) erreur ();
}
void affiche-carres (int d, int f)
{  int i ;
   for (i=d ; i<=f ; i++)
     printf ("%d a pour carre %d\n", i, i*i);  }
void erreur (void)
{  printf ("*** erreur ***\n"); }
```

Les fonctions

Lors de l'appel d'une fonction, les paramètres sont convertis automatiquement dans les types déclarés dans la définition de la fonction. Exemple:

```
....
int a = 200;   int resultat;
resultat = pow(a, 2);
...
```

A l'appel de la fonction *pow*:

- La valeur de a et la constante 2 sont converties en ***double***
- Le résultat retourné est converti en ***int*** avant d'être affecté à *resultat*.

Les fonctions

En C, les arguments sont transmis "par valeur":

```
main()
{
void echange (int a, int b);
int n=10, p=20;
printf ("avant appel : %d  %d\n",n, p);
echange (n, p);
printf ("après appel : %d  %d", n, p);
}
void echange (int a, int b)
{
int c ;
printf ("début échange : %d % d\n", a, b);
c = a; a = b; b = c;
printf ("fin échange : %d %d\n", a, b);
}
```

```
avant appel :     10  20
début échange :   10  20
fin échange :     20  10
après appel :     10  20
```


Les fonctions

Les arguments sont transmis "par valeur" ce mode se transmission interdit qu'une fonction modifie les arguments.

Mais on peut contourner ce problème:

- En utilisant des "**variables globales**"
- En transmettant la "**valeur" de "l'adresse d'une variable**".

➢ Si la transmission des arguments se fait "**par valeur**", les arguments effectifs peuvent être une expression.

➢ Si le mode de transmission est celui "par adresse", les arguments effectifs ne peuvent être qu'une ***lvalue***.

Les fonctions

Exemple d'utilisation d'une **variable globale**

```
#include <stdio.h>
int i;
main()
{
void optimist (void);
for (i=1 ; i<=5 ; i++)
optimist();
}
void optimist(void)
{
printf ("il fait beau %d fois\n", i);
}
```

```
il fait beau 1 fois
il fait beau 2 fois
il fait beau 3 fois
il fait beau 4 fois
il fait beau 5 fois
```

- Rien n'empêche la fonction ***optmist*** de modifier la valeur de ***i***.
- Cette façon de faire peut provoquer des erreurs si la variable est modifiée insidieusement par une fonction ...

Les fonctions

Transmission des adresses des arguments:

```
main()
{
void echange (int *a, int *b);
int n=10, p=20;
printf ("avant appel : %d  %d\n",n, p);
echange (&n, &p);
printf ("après appel : %d  %d", n, p);
}
void echange (int *a, int *b)
{
int c ;
printf ("début échange : %d % d\n", *a, *b);
c = *a; *a = *b; *b = c;
printf ("fin échange : %d %d\n", *a, *b);
}
```

```
avant appel :     10  20
début échange :   10  20
fin échange :     20  10
après appel :     20  10
```


Les fonctions

Lorsqu'un vecteur est passé en argument, c'est la valeur de l'adresse de son 1[er] élément qui est transmise. La fonction n'a, apriori, pas l'information de la dimension du tableau!

```
main()
{  int   tab[5]  =  {  1,  9,  10,  14,  18};
   int   somme(int  t[],  int  n);
   void  impression(int  *t,  int  n);
   printf("%d\n",  somme(tab, 5));
   impression(tab,  5);     }

int    somme(int  t[],  int  n)
{   int  i,  som=0;
   for  (i=0;  i  <  n;  i++)  som  +=  t[i];
   return  som;    }
```


Les fonctions

Une fonction peut avoir comme arguments des structures. Elle peut même retourner un tel objet.

```
typedef struct  { float  v[6]; }  Vecteur;
main()
{  Vecteur  vec  =  {  {1.34f,  8.78f,  10.f, 4.f,  22.12f,  3.145f}  };
   Vecteur  inv;
   Vecteur  inverse(  Vecteur  vecteur,  int  n  );
   int       i,  n  =   sizeof(vec.v) / sizeof (vec.v[0]);
   inv  =  inverse(  vec,  n  );
   for(  i=0;  i  <  n;  i++  )
        printf(  "inv.v[%d]  :  %f\n",  i,  inv.v[i]  );     }
Vecteur  inverse(  Vecteur  vecteur,  int  n  )
{   Vecteur  w; int    i;
    for(  i=0;  i  <  n;  i++  )
       w.v[i]  =  vecteur.v[i]  ?  1./vecteur.v[i]  :  0.f;
    return  w;   }
```


Les fonctions

Il convient d'être prudent lors de l'utilisation d'une fonction retournant un pointeur. Il faut éviter l'erreur qui consiste à retourner l'adresse d'une variable temporaire!

```
main()
{
char  *p;
char  *ini_car(void);
p  =  ini_car();
printf("%c\n",  *p);
}
char  *ini_car(void)
{
char  c  =  '#';
return(&c);        <===    ERREUR
}
```


Les fonctions

Le langage C autorise la **récursivité**, celle ci peut prendre deux aspects :

- Une fonction comporte dans sa définition au moins un appel a elle-même: récursivité directe
- L'appel d'une fonction entraîne celui d'une autre fonction qui appelle la fonction initiale: récursivité croisée

Un exemple classique (inefficace sur le plan du temps d'exécution)

```
long fact (int n)
{
if (n>1)  return (fact(n-1)*n);
     else return(1);
}
```

Entrées-sorties de haut niveau

Les entrées-sorties de haut niveau intègrent deux mécanismes distincts :

- le formatage des données
- la mémorisation des données dans une mémoire tampon

Toute opération d'entrée-sortie se fait par l'intermédiaire d'un flot de données (*stream*) qui est une structure de données de type ***FILE*** faisant référence à:

- la nature de l'entrée-sortie
- la mémoire tampon
- le fichier sur lequel elle porte
- la position courante dans le fichier
- ...

Le type ***FILE*** est défini dans ***stdio.h***

Entrées-sorties de haut niveau

Trois flots de données sont prédéfinis:

- stdin: en lecture sur l'entrée standard (clavier)
- stdout: en écriture sur la sortie standard (écran)
- stderr: en écriture sur la sortie erreur standard (écran)

La création d'un nouveau flot de données s'effectue par l'appel à la fonction ***fopen***

La fonction ***fclose*** permet de le fermer

Entrées-sorties de haut niveau

La fonction ***fopen,*** de type FILE*, ouvre un fichier et lui associe un flot.

Sa syntaxe est:

fopen("nom-de-fichier","mode");

- Le premier argument est le nom du fichier fourni sous forme d'une chaîne de caractères.
- Le second argument est une chaîne de caractères qui spécifie le mode d'accès au fichier.

Entrées-sorties de haut niveau

Les spécificateurs de mode d'accès diffèrent suivant le type de fichier considéré:

- les fichiers textes: les caractères de contrôle sont interprétés lors de la lecture et de l'écriture
- les fichiers binaires: permettent de transférer des données sans transcodage. Sont plus efficaces, mais les fichiers ne sont pas portables (le codage dépend des machines)

Le tableau suivant détaille les différents modes d'accès.

Entrées-sorties de haut niveau

"r"	ouverture d'un fichier texte en lecture
"w"	ouverture d'un fichier texte en écriture
"a"	ouverture d'un fichier texte en écriture à la fin
"rb"	ouverture d'un fichier binaire en lecture
"wb"	ouverture d'un fichier binaire en écriture
"ab"	ouverture d'un fichier binaire en écriture à la fin
"r+"	ouverture d'un fichier texte en lecture/écriture
"w+"	ouverture d'un fichier texte en lecture/écriture
"a+"	ouverture d'un fichier texte en lecture/écriture à la fin
"r+b"	ouverture d'un fichier binaire en lecture/écriture
"w+b"	ouverture d'un fichier binaire en lecture/écriture
"a+b"	ouverture d'un fichier binaire en lecture/écriture à la fin

Entrées-sorties de haut niveau

- La valeur retournée par ***fopen*** est un pointeur sur ***FILE***
- Si l'ouverture a réussi, le pointeur retourné permet de repérer le fichier, et devra être passée en paramètre à toutes les fonctions d'entrées-sorties sur le fichier
- Si l'ouverture est impossible, ***fopen*** renvoie la valeur ***NULL***

Conditions particulières et cas d'erreur

- Si le mode contient la lettre ***r***, le fichier doit exister!
- Si le mode contient la lettre ***w***, si le fichier n'existe pas, il est créé. Si le fichier existe déjà son ancien contenu est perdu.
- Si le mode contient la lettre ***a***, si le fichier n'existe pas, il est créé. Si le fichier existe déjà son ancien contenu est conservé. Les écritures se font à la fin du fichier au moment de l'exécution de l'ordre d'écriture.

Entrées-sorties de haut niveau

La fonction ***fclose*** permet de fermer le flot associé à un fichier créer par la fonction ***fopen***.

Sa syntaxe est:

fclose(flot) ;

Avec *flot* le pointeur de type ***FILE*** retourné par la fonction ***fopen*** correspondant.

fclose retourne un entier:

- zéro si l'opération s'est déroulée normalement
- -1 (EOF) en cas d'erreur

Entrées-sorties de haut niveau

La fonction d'écriture ***fprintf***, analogue à ***printf***, permet d'écrire des données dans un fichier.

Sa syntaxe est:

fprintf(flot, "chaîne de format",exp_1, ..., exp_n);

Où:

- flot est le flot de données retourné par la fonction ***fopen***
- Les spécifications de format utilisées pour la fonction ***fprintf*** sont les mêmes que pour ***printf***

Entrées-sorties de haut niveau

La fonction ***fscanf***, analogue à ***scanf***, permet de lire des données à partir d'un fichier.

Sa syntaxe est semblable à celle de ***scanf***

fscanf(flot, "chaîne de format ",argument-1,...,argument-n)

Où:

- flot est le flot de données retourné par la fonction ***fopen***
- Les spécifications de format utilisées pour la fonction ***fscanf*** sont les mêmes que pour ***scanf***

Entrées-sorties de haut niveau

Ressemblants à ***getchar*** et ***putchar***, ***fgetc*** et ***fputc*** permettent respectivement de lire et d'écrire un caractère dans un fichier.

La fonction ***fgetc*** retourne le caractère lu dans le fichier. Elle retourne ***EOF*** lorsque la fin du fichier est détectée .

int fgetc(FILE* flot);

La fonction ***fputc*** écrit caractère dans le flot de données et retourne l'entier correspondant au caractère écrit ou ***EOF*** en cas d'erreur):

int fputc(int caractere, FILE *flot);

il est toujours utile de tester la valeur lue ou écrite pour détecter la fin du fichier ou un problème d'écriture.

Entrées-sorties de haut niveau

Il existe également deux versions optimisées des fonctions ***fgetc*** et ***fputc*** qui sont implémentées par des macros.

Leur syntaxe est similaire à celle de ***fgetc*** et ***fputc*** :

int getc(FILE* flot);

int putc(int caractere, FILE *flot);

Entrées-sorties de haut niveau

Les fonctions d'entrées-sorties binaires transfèrent des données sans transcodage

Elles sont plus efficaces mais les fichiers produits ne sont pas portables puisque le codage des données dépend des machines

Elles sont utiles pour des données de grande taille ou renfermant un type composé

Entrées-sorties de haut niveau

Leurs prototypes sont:

size_t fread(void *pt, size_t taille, size_t nb, FILE *flot);

size_t fwrite(void *pt, size_t taille, size_t nb, FILE *flot);

Où:

pt: l'adresse de début des données à transférer
taille: la taille des objets à transférer
nb: leur nombre
size_t est un alias de ***unsigned int***

Elles retournent toutes deux le nombre de données transférées

Entrées-sorties de haut niveau

Les différentes fonctions d'entrées-sorties permettent d'accéder à un fichier en **mode séquentiel:** les données du fichier sont lues ou écrites les unes après les autres.

Il est possible d'accéder à un fichier **en mode direct:** en se positionnant à n'importe quel endroit du fichier.

Entrées-sorties de haut niveau

La fonction ***fseek*** permet de se positionner à un endroit précis:

int fseek(FILE *flot, long deplacement, int origine);

deplacement: détermine la nouvelle position dans le fichier. Il s'agit d'un déplacement relatif par rapport à l'origine ; il est compté en nombre d'***octets***.
origine peut prendre trois valeurs:
SEEK_SET (égale à 0) : début du fichier
SEEK_CUR (égale à 1) : position courante
SEEK_END (égale à 2) : fin du fichier

La fonction ***int rewind(FILE *flot);***

permet de se positionner au début du fichier. Elle est équivalente à ***fseek(flot, 0, SEEK_SET);***

Entrées-sorties de haut niveau

La fonction ***long ftell(FILE *flot);***

retourne la position courante dans le fichier (en nombre d'octets depuis l'origine).

Allocations dynamiques

Dans un programme, chaque variable occupe un certain nombre d'octets en mémoire réservés de manière automatique par les déclarations et le nombre d'octets était connu pendant la compilation.

Quand on travaille avec des données dont on ne peut pas prévoir la taille, on réserve toujours l'espace maximal prévisible

Pour éviter ce gaspillage on fait appel à la gestion dynamique de la mémoire lors de l'exécution du programme.

Allocations dynamiques

Exemple

Mémoriser 10 phrases saisies au clavier. Ne pouvant pas prévoir à l'avance le nombre d'octets à réserver pour les phrases, on déclare juste un tableau de 10 pointeurs sur des ***char*** par:

char *TEXTE[10];

Ces 10 pointeurs seront réservés automatiquement.....

La longueur des phrases n'étant connue que pendant l'exécution du programme, la réservation de la mémoire pour les 10 phrases se fera de manière dynamique.

Allocations dynamiques

Pour demander de la mémoire manuellement, on a besoin d'inclure la bibliothèque ***<stdlib.h>*** qui contient deux fonctions dont on a besoin:

- ***malloc*** : demande au système d'exploitation la permission d'utiliser de la mémoire
- ***free*** : permet d'indiquer à l'OS que l'on n'a plus besoin de la mémoire qu'on avait demandée

Trois étapes a suivre lors de l'allocation manuelle de mémoire:

1. appeler ***malloc*** pour demander de la mémoire
2. vérifier la valeur retournée par ***malloc*** pour savoir si on a réussi à allouer la mémoire
3. libérer avec ***free*** l'espace mémoire réservé quand on a fini de l'utiliser

Allocations dynamiques

La fonction ***malloc*** a pour prototype:

void* malloc(size_t nombreOctetsNecessaires);

Elle prend un paramètre: le nombre d'octets à réserver et renvoie un pointeur sans type ***(void*)***

Cette fonction renvoie un pointeur indiquant l'adresse réservée mais ne sait pas quel type de variable on cherche à créer

C'est le pointeur qui va recevoir la première adresse réservée qui définit le type créer

```
int* memoireAllouee = NULL;
memoireAllouee = malloc(sizeof(int));
```


Allocations dynamiques

Tester le pointeur!

malloc renvoie un pointeur l'adresse qui a été réservée.
Deux possibilités :

- si l'allocation a marché, le pointeur contient une adresse
- si l'allocation a échoué, le pointeur contient l'adresse NULL.

Il est peu probable qu'une allocation échoue, mais cela peut arriver.

Si le pointeur est différent de NULL, le programme peut continuer, sinon il faut afficher un message d'erreur ou même mettre fin au programme.

Allocations dynamiques

On utilise la fonction ***free*** pour libérer la mémoire dont on ne se sert plus. Son prototype est:

void free(void* pointeur);

Elle a juste besoin de l'adresse mémoire à libérer.

- La fonction ***free*** peut aboutir à un désastre si on essaie de libérer de la mémoire qui n'a pas été allouée par ***malloc***.
- La fonction ***free*** ne change pas le contenu du pointeur; il est conseillé d'affecter la valeur NULL au pointeur immédiatement après avoir libéré le bloc de mémoire qui y était attaché.
- Si nous ne libérons pas explicitement la mémoire à l'aide ***free***, alors elle est libérée automatiquement à la fin du programme

Allocations dynamiques

Autres fonction d'allocation dynamique de la mémoire:

void * calloc (size_t n, size_t t)

alloue ***n*** blocs de taille ***t***. elle est équivalente à ***malloc(n * t);*** mais avec la mémoire réservée est mise à 0

void * realloc (void * base , size_t t)

tente de redimensionner un bloc mémoire donné à la fonction par son adresse ***base*** avec une nouvelle taille ***t***

- Si une erreur se produit, la fonction retourne le pointeur NULL sinon la fonction renvoie un pointeur vers l'adresse du nouveau bloc réalloué.
- Le contenu de ce qui se trouvait dans le bloc d'origine est sauvegardé!